A Model-Based Approach to Managing Feature Binding Time in Software Product Line Engineering

Armaya'u Zango Umar
Lancaster University
Lancaster, United Kingdom
a.umar1@lancaster.ac.uk

Jaejoon Lee
Lancaster University
Lancaster, United Kingdom
j.lee3@lancaster.ac.uk

ABSTRACT

Software Product Line Engineering (SPLE) is a software reuse paradigm for developing software products, from managed reusable assets, based on analysis of commonality and variability (C & V) of a product line. Many approaches of SPLE use a feature as a key abstraction to capture the C&V. Recently, there have been increasing demands for the provision of flexibility about not only the variability of features but also the variability of when features should be selected (i.e., variability on feature binding times). Current approaches to support variations of feature binding time mostly focused on ad hoc implementation mechanisms. In this paper, we first identify the challenges of feature binding time management and then propose an approach to analyze the variation of feature binding times and use the results to specify model-based architectural components for the product line. Based on the specification, components implementing variable features are parameterized with the binding times and the source codes for the components and the connection between them are generated.

KEYWORDS

Model-Based Software Product Line; Models and Components; Product Line of Product Lines; Variable Binding Time; Feature Binding Time

1 INTRODUCTION

Software Product Line Engineering (SPLE) is a software reuse paradigm for developing software products from managed reusable assets. The reusable assets are engineered based on analysis of *commonality* and *variability* (C & V) of a family of software products in a specific problem area, known as a *domain*. The approach uses a *feature* as a key abstraction [16, 19, 26]. The commonality represents the set of mandatory features, the shared traits, which manifest in all the products of the product line. On the other hand, a feature in the variability category can be one of the optional/alternative features or forms a part of an inclusive OR group. Composition rules are also used to constrain the selection of features in the variability category. A feature model[20] is widely used to present C & V information of a product line compactly(see Fig.1 for example). Each product in the product line is derived from a selection of a valid combination of features —a process known as *product configuration*. The phase, in the product life cycle, of which a feature is selected and bound to a product is known as *feature binding time*[31, 33]. Engineers used the results of C & V analyses to design a Product Line Architecture (PLA) - a generic software architecture that can be tailored to produce a potentially unique architecture for each of the products of a product line.

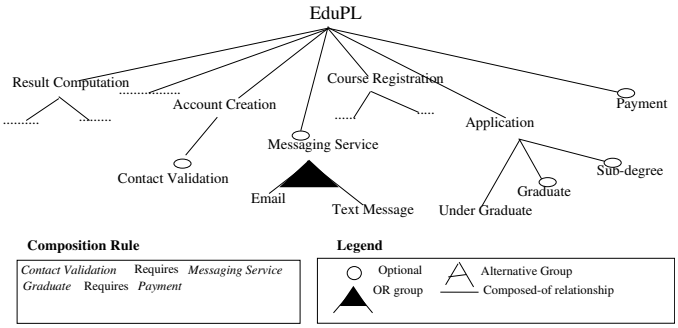

Figure 1: Partial feature model of EduPl

1.1 Background

Recently there have been increasing demands for flexibility of not only the variability of features but also the variability of when features should be selected (i.e., variability of feature binding time). This is often the case when products of a product line have to be configured for delivery to multiple product lines[2, 13]. Fig.1 presents an example of enterprise software for tertiary institutions of an anonymous country. The product line, referred to as Educational Product Line (EduPl), was initiated by our partner university in that country. The vision of the product line is to provide software products to sister universities, other higher institutions, and Enterprise Resource Planning (ERP) vendors. The educational institutions in the country have common and centralized regulatory agencies - which make their core operations largely the same- hence a product line. In this example, a customer institution placing a direct order, through their IT department, requires the binding of *Messaging Service* and *Contact Validation* to be decided at product configuration time. Other customers, who are ERP vendors, require the same *Messaging Service* and *Contact Validation* to be decided at installation time after their salesman negotiates with customers over the product price. Yet still, customers hitherto deselected a feature may demand the same feature at operation time when their services evolved. For example, customer institution starting graduate courses should select and bind (to be downloaded or made available in a memory stick) the *Graduate* feature to their product already in operation. As such, the product line engineers of EduPl should develop the core assets to support the different binding times of these features.

The above example exposes three distinctive challenges: 1) how to identify the set of features that should be selectable at different binding times; 2) how to engineer and instantiate reusable assets to support such selections; and 3) how to maintain the feature dependencies (e.g., require and mutual exclusion) throughout the different phase (e.g., product configuration time, installation time, or operation time) of feature selection. Without addressing these challenges, the product line assets have to go through an ad hoc adaptation to serve requests for varying binding times [2].

Ad-hoc adaptations of binding times, through hacks, can induce architectural *degradation* [32], which affects maintainability and adaptability. Also, it can slow-down the *time-to-market*, increase production costs, and affect the quality of the products. Equally sub-optimal is to develop separate assets for the different categories of customers; i.e., splitting, supposedly, a single asset-based into a set of parallel asset-based. This approach, clearly adds up to production and maintenance costs because a product line company has to maintain multiple assets for the different categories of customers.

The proliferation of software provider-consumer relationships has aggravated the demands for flexibility in the selection of features at different times. The provider is a product line company that has to deliver partially configured products to consumers that are separate product lines [13]. In the literature, the terms Product Line of Product Line (PoPs) and Multi-Product Lines (MPLs) are used to depict the software provider-consumer relationship as a form of associations between product lines of separate logical boundaries [6, 14, 17, 30]. Hartman *et al* [11, 12] used the term Software Supply Network (SSN) to qualify the associations between the multiple product lines as a network of independent stakeholders engaged in a software supplier-customer relationship.

There have been attempts to address the challenge of providing flexible feature binding time [4, 7, 8, 27, 27, 34]. However, some of these approaches addressed flexibility at a fine granular level (e.g., flexible composition of low-level model elements such as states and transitions in state diagram)[34]. Others proposed some ad hoc manipulations that are only tenable in some specific development environments [8]. All in all, a comprehensive approach to address the flexibility of feature binding time selection is largely absent.

In this paper, we describe our initial ideas for supporting flexible feature binding time in SPLE. We first elaborate the challenges to set the context and provide the overview of the approach with examples.

1.2 Challenges for Varying binding times

In the following we elaborate more on the three challenges we have identified with examples:

C1 How to proactively discover features whose binding time need to vary and use this knowledge in the design of reusable assets. By 'proactive' we mean that we should check with the potential customers of the product line to find out binding time requirements before developing the core assets. Identifying different binding time requirements entails careful analysis on where and how the customers would use the product (i.e., the usage contexts of the customers).

C2 How to develop product line assets to support the different binding times. The challenge is not only about supporting different binding times for the different features but also about supporting different binding time of the same feature and at the same variation point. A variation point is a place where different variants can be bound for different product configurations[16].

C3 How to maintain consistency between feature selection and binding time instantiation. For example, when a parent feature is selected with a late binding time, its child features should all be selected at not earlier than the parent feature's binding time and we should make sure there are no other features that require one of the child features at an earlier binding time. Supporting flexibility in the feature binding time requires a robust management of feature dependencies [18] with their different binding time.

To address the above challenges, we propose an approach to manage the variations of feature binding time. Our work is based on feature binding unit analysis - an approach initially proposed to address dynamic reconfiguration of software systems in a dynamic software product line (DSPL) [24]. Specifically, we adapted feature binding unit analysis as a guideline on how to decompose product line architecture to match the boundaries of features that must be bound together. We extend this approach with flexible connection mechanisms between components of separate binding units. We also propose a means to instantiate product line assets with the different binding time. Fig.2 shows the activities and work products of the proposed approach, which are explained in the following section.

2 OVERVIEW OF THE APROACH

To illustrate the approach, the following elaborates on some of the features in Fig.1: The *Account Creation* feature is to create an account with the institution's portal. The system can optionally validate an applicant's contact using *Contact Validation*. The *Contact Validation* feature requires *Messaging Service* to send randomly generated token to the applicant's email or a Short Message Service (SMS) to the applicant's mobile telephone. *Email* and *Text Message* are inclusive OR features and are the specialization of *Messaging Service*. An institution could optionally charge application fees through the *Payment* feature. There are different types of applications: *Undergraduate*, Graduate and *Sub-degree*. Both *Graduate* and *Sub-degree* require *Payment*.

2.1 Process Activities

SPLE consists of two major engineering processes: Domain Engineering (DE) and Application Engineering (AE). DE is the process of developing assets for reuse while AE is the process of deriving a specific product from the reusable assets. The following activities fall within the two engineering processes.

2.1.1 Feature Modeling. In DE process, feature modeling kick-starts the engineering activities of which a feature model is developed. Detailed on feature modeling activity can be found in [19].

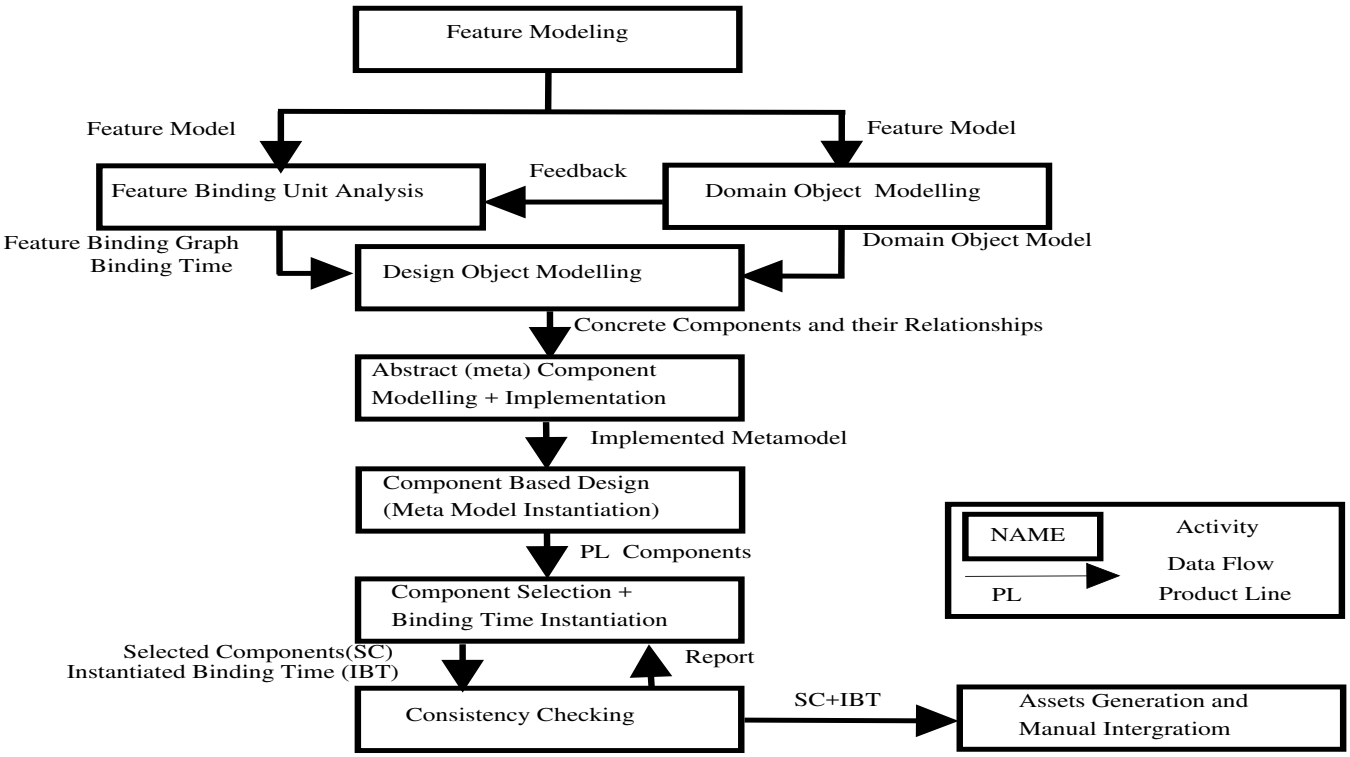

Figure 2: A proposed framework for managing feature binding time

2.1.2 Feature Binding Unit Analysis . To engineer the core assets to support different binding time of features, the indivisible binding units must be established. As depicted in Fig.2, a feature model serves as input to binding unit analysis. A feature binding unit (FBU) is a set of features that share the same binding time (all the features in the same binding unit must be bound together for the correct behavior of the product)[24]. Feature binding unit identification starts with the identification of a service feature. Each service feature represents a major functionality of the system that may be added/removed as a service unit. In EduPL(Fig.3), *Account Creation* and *Application* are examples of service features. Beginning from the service feature, an engineer should identify the constituents of a binding unit by traversing the feature model along the feature relationships and its composition rules (i.e., cross-tree constraints such as require/mutual exclusion). Each binding unit is assigned a name, in a capital letter, similar to the name of the major feature in the unit. For example, ACCOUNT and VALIDATION are the names assigned to the binding units containing *Account Creation* and *Contact Validation* respectively (see Fig.3).

In our approach, in addition to analysis of C & V of the product features, we also analyze C & V of the different product usage contexts [12, 22, 25]. We aim to capture the binding time of features in terms of product lifecycle (i.e., when and how each of the feature binding happens), by examining different usage contexts of the products and in anticipation of product evolution. There are many recognized binding time of features [31, 33]. In this paper, we focus on three binding times that are most visible to the

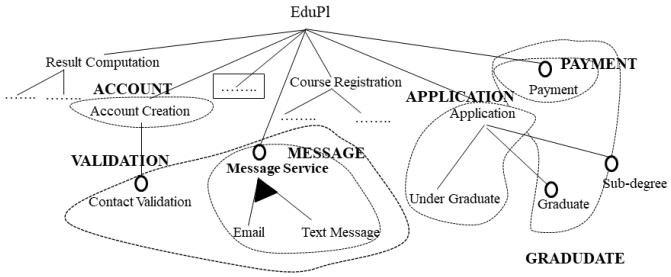

Figure 3: Feature binding unit identification from EduPL feature model

product line customer (i.e. when a feature is included in a product configuration and made available to a customer): (i) *Product configuration (order) time(PT)* (ii) *Installation (deployment) time (IT)* and (iii) *Operation time (OT)*. The output of feature binding analysis is a binding unit graph (see the upper part of Fig.4). In Fig. 4, the dotted lines represent a variable binding time. For example, binding between VALIDATION and ACCOUNT could happen at either product configuration time, installation time, or even at operation time.

2.1.3 Domain Object Modeling. A feature model and other domain analyses (e.g. noun phrase analysis) are also inputs to a Domain Object Modeling activity (see Fig.2). A domain object model is

a product line's model of conceptual classes or objects and their relationships; it is an input to Design Object Model. Detail of domain object modeling activity can be found in [26].

2.1.4 Design Object Modeling. A design object model is an embodiment of functionalities required for the product line[26]. In the design object model, a domain engineer captures concrete domain design, in terms of object based components and their responsibilities that can be mapped to the identified binding units. As shown in Fig.2, the feature binding graph and the binding time information are important inputs for organizing design object model. Our approach is not limited to Object-Oriented(OO) component, it can be used with non-OO architecture description approaches. For the variation of binding time to be feasible, the mapping between binding units and the variation points in the design object model must be established explicitly. In Fig. 4, the mapping between binding units (depicted as the bubble with the optional or the alternative icon) and the corresponding variation points identified in the object model (denoted by the filled rectangle) is depicted with dotted arrows.

2.1.5 Abstract (meta) Component Modeling and Implementation. Following Model-Driven approach [28], the captured components and their relationship (see the bottom part of Fig. 4) must be described in abstract terms (i.e. Metamodeling). A metamodel defines the possible constructs of a model in terms of abstract syntax and static semantic [28]. The purpose is to enable machine processing of the model generically (i.e. Metaprogramming). The metamodel has to be implemented in some sort of infrastructure. Fig.5 depicts a metamodel of the EduPL in Ecore. Ecore is a metamodeling language that is part of Eclipse Modeling Framework (EMF)[1] infrastructure.

The metamodel in Fig.5 has *Component* as its metaclass which is a generalization over the component type of the intending model (the EduPl architectural model in this case). The specialized metaclasses are *Service*, *Data*, *View* and *Control* components. A Service component carries out computing services or functions[35], for example, the generation of random token for user's contact validation in the implementation of the *Contact Validation* feature. A view component receives input from the end user and displays output from the system (e.g. an application form). A control component controls or coordinates other components[35]; as such it controls the behavior of the system or subsystem. A data component represents an entity that can be saved and retrieved from a permanent storage (e.g. an applicant information). The discrimination between component types is useful for determining a suitable connection mechanism between the different type of components and also for imposing architectural constraints (e.g. a view component should not access data component directly).

Notice from Fig.5, we elevate *Port* and *Connector* with first-class status in the metamodel, in contrast to their usual relegation to the second-class elements at the implementation level. In essence, we abstract over concrete ports and connector by generating them and configuration scripts (as the connection mechanism) based on the instantiated binding time. For example, if the bindings of *ValidationManager* and *MessageHandler* have to be decided at product configuration time, we can generate procedure call ports and tightly coupled connector between these components and the *AccountCreationController* component. If, however, the bindings of *ValidationManager* and *MessageHandler* have to be decided at installation time, we may generate event ports, event connector, and configuration scripts to remove the dependency between the components. In this way, we make it possible to include/exclude the *ValidationManager* and *MessageHandler* components from the product configuration at installation time. In the case of operation time binding, we should generate message ports, connectors, and configuration scripts to make it possible to compile the components separately and include/exclude them at operation time.

2.1.6 Component Based Design (Metamodel Instantiation). The component-based product line architecture is then modeled as instances of the implemented metamodel. A mapping has to be established between the instantiated components and the features they implement in the feature model. Deselecting a feature from the feature model removes the corresponding model component. We assume the existence of the mapping tool such as the commercial feature model connector of Pure Variant[2] or the open-source tool developed by Czarnecki *et al* [5]. Those tools embed a model repair algorithm to fix an ill-formed model automatically. For example, the domain engineer should design concrete components to correspond to the design object model depicted at the bottom of Fig.4. The engineer should also mark *ValidationManager* and *MessageHandler* as variable components and their binding will be determined by the application engineer in the application engineering process.

2.1.7 Component Selection and Binding Time Instantiation. During Application Engineering process, the application engineer selects features from the feature model, which triggers the removal of components mapped to the deselected feature. The engineer also instantiates the required binding time of features on the components whose features are selected.

2.1.8 Consistency Checking. Binding time instantiation triggers the consistency checking process. In case of an error, the application engineer may have to change feature selections or binding time selections.

2.1.9 Assets Generation and Manual Code Integration. We generate the skeleton implementation from the architectural components. Based on the instantiated binding time, we also generate concrete ports and the connectors for the components. We do not intend to generate every implementation artifacts from the model as we believe the separation of high-level architectural concerns and low-level implementation is useful for our approach to scale in practice.

3 RELATED WORK

Traditionally, usage context analyses [12, 25] are used to discover various operational contexts of products of the product line. Goal-driven analysis [22] is also used to expose the coverage of broad objectives of which products of a product line were meant to achieve.

[1] https://www.eclipse.org/modeling/emf/

[2] https://www.pure-systems.com/products/extensions/pure-variants-connector-for-emf-feature-mapping-307.html

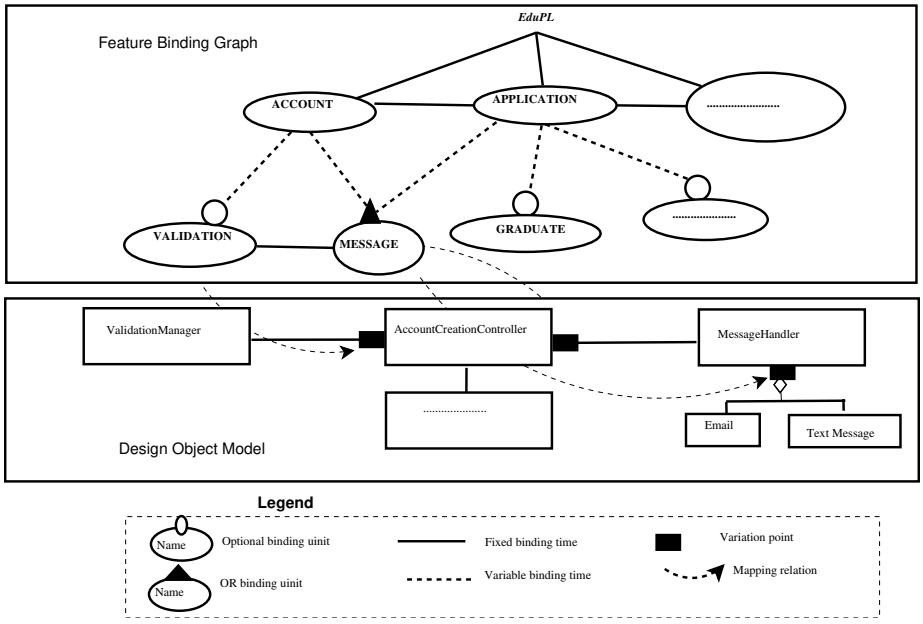

Figure 4: A mapping between binding units in the binding unit graph and variation points in the design object model adapted from *Lee et al (2004)*

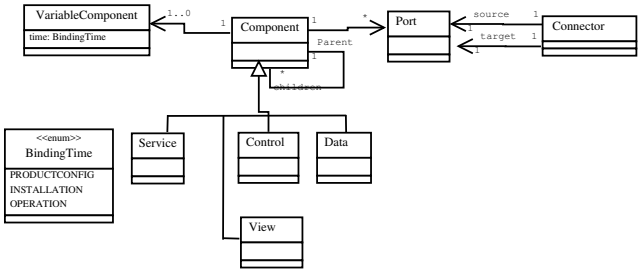

Figure 5: Metamodel of EduPL component architecture

These analyses expose the implications of goals and contexts on the selection of, mainly non-functional, features of the product line but have not considered the variations of binding time between the different goals and contexts.

In the implementation, Chakravarthy *et al* [4] proposed a combination of design patterns and Aspect-Oriented Programming (AOP) to achieve binding time flexibility. In their approach, the same feature is implemented as different *aspects*, each *aspect* for one target binding time at the same variation point. This approach can be useful if the variation is fixed and features can be bound to stable variation interfaces. However, the approach limits reusability because multiple implementations of features have to be maintained. Dolstra *et al* [8] discussed how developers manipulated a combination of pre-processor, compiler, and linker to achieve binding time flexibility in the retrofitted Linux kernel. The approach described in [8] is ad-hoc and only attainable if the development environment supports separate building and linking of modules.

Most researches in binding time management fall within a research theme of DSPL [1, 10, 15, 21, 23] and have an explicit focus on reconfigurations of features at runtime, often based on changes in the execution context of a software product; they largely ignore pre-runtime variation. DSPL is effective when most of the variations exist only when the product is in operation (e.g., in robotic domain)[9, 21, 29]. In contrast, in some domains such as enterprise software, management of variations at different phases of a software lifecycle, the focus of this paper, is non-trivial. Specifically, this paper is about supporting flexibility of variations at different phases of a software lifecycle.

Model-driven approaches have the potential to manage variations by mapping features in the feature model to model elements [3, 5]; i.e. model elements are annotated with a presence condition of features from the feature model. For example, In [5], a feature from the feature model can be mapped to class diagram and activity diagram via annotations. These approaches include support for enforced variations as well consistent product derivation. However, their support to addressing variations of binding time [27, 34], focus on fine granular variations which make these approaches difficult to scale. In contrast, we emphasize modeling coarse-grained components that are critical to the architecture of the product line.

In summary, managing variations of feature binding time need to be further investigated

4 CONCLUSION AND FUTURE WORK

We identified the challenges of managing feature binding time in SPLE and sketch our idea to address them. Our approach is a generative framework where variable components are parameterized

with three possible binding times (product configuration time, installation time and operation time). Assets are generated based on the binding time that has been instantiated during application engineering process. For future works, we shall define concepts of connection mechanisms that are platform/middleware agnostic and the first transformation (after passing consistency checking) should generate a model with those concepts. Subsequent transformations should then be defined for each of the target platform/middleware. In model-driven parlance, we aim at multistage transformations from the Platform Independent Model (PIM) to Platform Specific Models (PSMs). For the consistency checking, we shall either defined consistency rules in Object Constraint Language (OCL)[3] or transform the PIM model into another form that allows its properties to be checked (e.g. using Theorem prover).

ACKNOWLEDGEMENT

This work is funded by National Information Technology Development Agency (NITDA)-Nigeria.

REFERENCES

[1] L. Baresi, S. Guinea, and L. Pasquale. 2012. Service-Oriented Dynamic Software Product Lines. *Computer* 45, 10 (Oct 2012), 42–48. https://doi.org/10.1109/MC.2012.289

[2] Jan BoschPetra and Bosch-Sijtsema. 2014. ESAO: A holistic ecosystem-driven analysis model. In *Software Business. Towards Continuous Value Delivery(ICSOB 2014)*, C. Lassenius and K. Smolander (Eds.). Lecture Notes in Business Information Processing LNBP, Vol. 182. Springer, Cham, Paphos, Cyprus, 179–193. https://doi.org/10.1007/978-3-319-08738-2_13

[3] Thomas Buchmann and Bernhard Westfechtel. 2014. Mapping feature models onto domain models: ensuring consistency of configured domain models. *Software and Systems Modeling* 13, 4 (2014), 1495—1527. https://doi.org/10.1007/s10270-012-0305-5

[4] Venkat Chakravarthy, John Regehr, and Eric Eide. 2008. Edicts: Implementing Features with Flexible Binding Times. In *Proceedings of the 7th international conference on Aspect-oriented software development (AOSD '08)*. ACM, Brussels, Belgium, 108–119. https://doi.org/10.1145/1353482.1353496

[5] Krzysztof Czarnecki and Michal Antkiewicz. 2005. Mapping features to models: A template approach based on superimposed variants. In *GPCE 2005: International Conference on Generative Programming and Component Engineering (Lecture Notes in Computer Science)*, Vol. 3676. 422—437. https://doi.org/10.1007/11561347_28

[6] Ferruccio Damiani, Ina Schaefer, and Tim Winkelmann. 2014. Delta-oriented multi software product lines. In *(SPLC'14)Proceedings of the 18th International Software Product Line Conference*. ACM, Florence, Italy, 232–236.

[7] A Van der Hoek. 2004. Design-time product line architectures for any-time variability. *Science of computer programming* 53, 3 (December 2004), 285–304.

[8] Eelco Dolstra, Gert Florijn, and Eelco Visser. 2003. *Timeline Variability: The Variability of Binding Time of Variation Points*. Technical Report UU-CS-2003-052. Reseach Institute of Computer Science and Mathematics, University of Groningen, Utrecht University, Groningen, The Netherlands.

[9] L. Gherardi, D. Hunziker, and G. Mohanarajah. 2014. A Software Product Line Approach for Configuring Cloud Robotics Applications. In *2014 IEEE 7th International Conference on Cloud Computing*. 745–752. https://doi.org/10.1109/CLOUD.2014.104

[10] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (April 2008), 93–95. https://doi.org/10.1109/MC.2008.123

[11] Herman Hartmann, Aart Matsinger, and Tim Trew. 2009. Supplier independent feature modelling. In *Proceedings of the 13th International Software Product Line Conference*. ACM, San Francisco, California, USA, 191–200.

[12] Herman Hartmann and Tim Trew. 2008. Using Feature diagrams with Context Variability to model Multiple Product Lines for Software Supply Chains. In *12th International Software Product Line Conference*. IEEE, Limerick, Ireland, 12–21. https://doi.org/10.1109/SPLC.2008.15

[13] Herman Hartmanna, Mila Keren, Aart Matsinger, Julia Rubin, Tim Trewa, and Tali Yatzkar-Hahamb. 2013. Using MDA for integration of heterogeneous components in software supply chains. *Science of Computer Programming* 78, 12 (December 2013), 2313–2330.

[14] G Holl, P Grünbacher, and R Rabiser. 2012. A systematic review and an expert survey on capabilities supporting multi product lines. *Information and Software Technology* 54, 8 (August 2012), 828–852.

[15] P. Istoan, G. Nain, G. Perrouin, and J. Jezequel. 2009. Dynamic Software Product Lines for Service-Based Systems. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, Vol. 2. 193–198. https://doi.org/10.1109/CIT.2009.54

[16] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering* 5, 1 (December 1998), 143–168.

[17] Niloofar Khedri and Ramtin Khosravi. 2015. Towards managing data variability in multi product lines. In *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*. IEEE, 1–8.

[18] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is there a mismatch between real-world feature models and product-line research?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. 291–302. https://doi.org/10.1145/3106237.3106252

[19] KwanwooLee, Kyo C.Kang, and Jaejoon Lee. 2002. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *Software Reuse: Methods, Techniques, and Tools*, Cristina Gacek (Ed.). Lecture Notes in Computer Science, Vol. 2319. Springer-Verlag, Berlin Heidelberg, 62–77.

[20] James A. Hess William E. Novak Kyo C. Kang, Sholom G. Cohen and A. Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-21. Carnegie-Mellon University, Pennsylvania , USA.

[21] Jaejoon Lee and K. C. Kang. 2006. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *10th International Software Product Line Conference (SPLC'06)*. 10 pp.–140. https://doi.org/10.1109/SPLINE.2006.1691585

[22] Jaejoon Lee, Kyo C. Kang, Pete Sawyer, and Hyesun Lee. 2014. A holistic approach to feature modeling for product line requirements engineering. *Requirements Engineering in Software Product Lines* 19, 4 (November 2014), 377–395.

[23] J. Lee, G. Kotonya, and D. Robinson. 2012. Engineering Service-Based Dynamic Software Product Lines. *Computer* 45, 10 (Oct 2012), 49–55. https://doi.org/10.1109/MC.2012.284

[24] Jaejoon Lee and Dirk Muthig. 2009. Feature Oriented Analysis and Design for Dynamically Reconfigurable Product Lines. In *Applied Software Product Line Engineering*, Sooyong Park Kyo C. Kang, Vijayan Sugumaran (Ed.). Auerbach, New York, 315.

[25] Kwanwoo Lee and Kyo C. Kang. 2010. Usage Context as Key Driver for Feature Selection. In *Software Product Lines: going Beyond*, Jan Bosch and Jaejoon Lee (Eds.). Lecture Notes in Computer Science, Vol. 6287. Springer-Verlag, Berlin Heidelberg, 32–46.

[26] Kwanwoo Lee, Kyo C. Kang, Wonsuk Chae, and Byoung Wook Choi. 2000. Feature-based approach to object-oriented engineering of applications for reuse. *Software – Practice Experience* 30, 9 (July 2000), 1025 –1046.

[27] Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. 2011. Unifying design and runtime software adaptation using aspect models. *Science of Computer Programming* 76, 12 (December 2011), 1247–1260. https://doi.org/10.1016/j.scico.2010.12.005

[28] Dimitrios S.Kolovos Richard F.Paige and Fiona A.C.Polack. 2014. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming* 96, 4 (December 2014), 396–416.

[29] C. Schlegel, T. Hassler, A. Lotz, and A. Steck. 2009. Robotic software systems: From code-driven to model-driven designs. In *2009 International Conference on Advanced Robotics*. 1–8.

[30] Reimar Schröter, Norbert Siegmund, and Thomas Thüm. 2013. Towards modular analysis of multi product lines. In *Proceedings of the 17th International Software Product Line Conference co-located workshops on - SPLC '13 Workshops*. Tokyo, Japan, 96–99.

[31] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. 2005. A taxonomy of variability realization techniques. *Software: Practice and Experience* 35, 8 (July 2005), 705–754. https://doi.org/10.1002/spe.652

[32] Richard N. Tylor, Nenad Medvidovic', and Eric M. Dashofy. 2010. *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Son Sons Inc.

[33] J. van Gurp, J. Bosch, and M. Svahnberg. 2001. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*. IEEE, 45—54. https://doi.org/10.1109/WICSA.2001.948406

[34] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira, and João Araújo. 2009. MATA: A unified approach for composing UML aspect models based on graph transformation. *Transactions on Aspect-Oriented Software Development VI* 5560 (2009), 191–237. https://doi.org/10.1007/978-3-642-03764-1_6

[35] Rebecca Wirfs-Brock and Alan McKean. 2002. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education, Greenwich, CT, USA.

[3] https://www.omg.org/spec/OCL/About-OCL/